Folder Organization

1. Philosophy: An Architecture That Breathes

The structure of our application is a direct reflection of our design philosophy. A clean, predictable, and scalable folder structure is essential for building applications that are a pleasure to maintain. It reduces cognitive load, streamlines collaboration, and ensures every developer knows exactly where to find what they need and where to put new code.

This guide defines the official, opinionated folder structure for all Play+ applications, based on the actual implementation of the play-angular-seed project.

Core Principles

- Centralized System Layer: All core Play+ helpers are initialized and exported from a top-level /system directory. All core files are prefixed with play. to create a clear visual distinction.
- Centralized Configuration: All overridable config files for the Play+ helpers are housed under /config and follow the play.*.config.json convention.
- Separation of Concerns: Clear boundaries are drawn between features, UI components, configurations, utilities, and foundational system services.
- Co-located Testing: Each test file lives alongside the file it tests, encouraging maintenance and discoverability.

2. Angular Folder Structure (Implemented)

This structure reflects the actual implementation of the play-angular-seed project, optimized for Angular applications built using Angular CLI with comprehensive Play+ integration.

4. Key Directory Explanations

Directory	Purpose & Guidelines
docs/	Comprehensive project documentation including README files for each Play+ helper system (accessibility, caching, error handling, etc.).
config/	User-overridable Play+ config files. Uses consistent play.*.config.json naming for clarity. Includes configurations for all Play+ systems.
components/	Pure presentational components (UI-only, no business logic). Examples: Button , ProductSearch , UserList . These are generic and reusable across features.
features/	Feature-sliced architecture. Each folder represents a feature (e.g., auth/). Encapsulates business logic, views, and state for that domain.
directives/	Custom Angular directives that extend HTML functionality. Examples: accessibility.directive.ts , feature-flag.directive.ts .
pipes/	Custom Angular pipes for data transformation. Examples: accessibility.pipe.ts , performance.pipe.ts .
guards/	Route guards for navigation control. Examples: auth.guard.ts , play-feature.guard.ts .
services/	Application-specific services that integrate with Play+ systems. Examples: auth.service.ts , product.service.ts .
core/	Core logic and singleton services. Includes global error handlers and interceptors.

Directory	Purpose & Guidelines
lib/	Shared utility functions—formatters, validators, math utilities, etc.
system/	Core Play+ system files. Houses reusable helpers (e.g., play.error.ts) and adapters (e.g., api.service.ts). Do not place app logic here.
types/	Shared TypeScript interfaces, enums, and models used across multiple layers.
middleware/	Guard functions, HTTP interceptors, authorization logic, etc.
reports/	Git-ignored outputs from audits (e.g., Lighthouse), code coverage, linting reports, etc.
styles/	Theming and global design tokens. Includes themes/ folder for modular theme files.
testing/	Testing utilities and helpers specific to the application.
scripts/	Build and utility scripts for development workflow automation.

5. Play+ System Integration

System Layer (/system)

The system layer contains all core Play+ helpers and services:

- API Integration : api.proxy.ts , api.routes.ts , api.service.ts
- Caching : play.cache.ts , play.cache.service.ts
- Error Handling : play.error.ts , play.error.service.ts
- Feature Flags : play.feature.ts , play.feature.service.ts
- Logging : play.log.ts , play.log.service.ts
- Performance : play.perf.ts , play.perf.service.ts

- Accessibility : play.a11y.ts , play.a11y.service.ts
- Security : play.security.ts , play.guard.service.ts
- Environment : play.env.ts , play.env.service.ts

Configuration Layer (/config)

All Play+ systems have corresponding configuration files:

- play.a11y.config.json - Accessibility settings
- play.api.config.json - API configuration
- play.cache.config.json - Caching behavior
- play.error.config.json - Error handling rules
- play.feature.config.json - Feature flag settings
- play.guard.config.json - Route guard configuration
- play.log.config.json - Logging preferences
- play.perf.config.json - Performance monitoring
- play.performance.config.json - Performance budgets
- play.security.config.json - Security policies

Application Integration

The application layer integrates with Play+ systems through:

- Services : Application-specific services in /src/app/services/
- Directives : Custom directives in /src/app/directives/
- Pipes : Data transformation pipes in /src/app/pipes/
- Guards : Route protection in /src/app/guards/
- Core : Global handlers and interceptors in /src/app/core/

6. Test Strategy

Play+ projects follow a test co-location strategy. This means that each test file lives directly alongside the file it tests.

Example (Angular)

Testing Infrastructure

The project includes comprehensive testing setup:

- Karma/Jasmine : Angular testing framework
- Test Scripts : playtest , playtest:watch , playtest:gen
- Coverage : Code coverage reporting
- Testing Utils : /src/app/testing/ for shared testing utilities

Benefits:

- Encourages test creation during development
- Makes test discovery intuitive
- Keeps test logic scoped and relevant
- Provides comprehensive coverage reporting

7. Build and Development Scripts

The project includes comprehensive npm scripts for development workflow:

Testing Scripts

- playtest - Run tests with coverage
- playtest:watch - Run tests in watch mode
- playtest:gen - Generate test files

Linting Scripts

- playlint - Run ESLint
- playlint:fix - Fix ESLint issues
- playlint:report:html - Generate HTML lint report

Performance Scripts

- perf:monitor - Bundle analysis
- perf:lighthouse - Lighthouse performance audit
- perf:lighthouse:html - HTML performance report
- perf:budget - Performance budget checking

Formatting Scripts

- format - Format code with Prettier
- format:check - Check code formatting

Report Serving

- serve:lint-report - Serve linting reports
- serve:perf-report - Serve performance reports

9. Development Workflow

Pre-commit Hooks

- Husky : Pre-commit hooks for linting and testing
- ESLint : Code quality enforcement
- Prettier : Code formatting consistency

Code Quality

- ESLint : Comprehensive linting rules
- Custom Rules : eslint-rules/play-state-rules.js
- Prettier : Consistent code formatting
- TypeScript : Strict type checking

Performance Monitoring

- Lighthouse CI : Automated performance audits
- Bundle Analysis : Webpack bundle analyzer
- Performance Budgets : Automated budget checking

Reporting

- Linting Reports : HTML reports for code quality
- Performance Reports : Lighthouse performance reports
- Coverage Reports : Test coverage analysis

Summary

The Play+ folder structure is designed for clarity, consistency, and maintainability. It embodies our design philosophy by drawing strong lines between app logic, reusable elements, and core system capabilities.

Key Features of the Implemented Structure:

- Centralized System Layer : All Play+ helpers in /system/
- Comprehensive Configuration : All configs in /config/
- Feature-Based Organization : Clear separation of concerns

- Testing Co-location : Tests alongside source files
- Documentation-First : Comprehensive docs in /docs/
- Performance Monitoring : Built-in performance tooling
- Code Quality : Automated linting and formatting
- Reporting : HTML reports for all audits

This guide should be treated as the canonical structure across all Play+ applications, enabling seamless onboarding, developer velocity, and architectural integrity.

The structure breathes with your application, growing and adapting as your needs evolve while maintaining the core principles that make Play+ applications a joy to work with.